Extending SPARQL algebra to support efficient evaluation of top-k SPARQL queries

Bozzon, A.; Della Valle, E; Magliacane, S.

published in
Search Computing
2012

Extending SPARQL Algebra to Support Efficient Evaluation of Top-K SPARQL Queries

Alessandro Bozzon[1], Emanuele Della Valle[1], and Sara Magliacane[1,2]

[1] Politecnico of Milano, P.za L. Da Vinci, 32. I-20133 Milano - Italy
[2] VU University Amsterdam, The Netherlands

Abstract. With the widespread adoption of Linked Data, the efficient processing of SPARQL queries gains importance. A crucial category of queries that is prone to optimization is "top-k" queries, i.e. queries returning the top k results ordered by a specified ranking function. Top-k queries can be expressed in SPARQL by appending to a SELECT query the ORDER BY and LIMIT clauses, which impose a sorting order on the result set, and limit the number of results. However, the ORDER BY and LIMIT clauses in SPARQL algebra are result modifiers, i.e. their evaluation is performed only after the evaluation of the other query clauses. The evaluation of ORDER BY and LIMIT clauses in SPARQL engines typically requires the process of all the matching solutions (possibly thousands), followed by a monolithically computation of the ranking function for each solution, even if only a limited number (e.g. $K = 10$) of them were requested, thus leading to poor performance.
In this paper, we present $\mathcal{S}$PARQL-$\mathcal{R}$ANK, an extension of the SPARQL algebra and execution model that supports ranking as a first-class SPARQL construct. The new algebra and execution model allow for splitting the ranking function and interleaving it with other operations. We also provide a prototypal open source implementation of $\mathcal{S}$PARQL-$\mathcal{R}$ANK based on ARQ, and we carry out a series of preliminary experiments.

1 Introduction

SPARQL [16] is a W3C recommendation that specifies a query language as well as a protocol for Linked Data (LD). An ever-increasing number of SPARQL endpoints allows to query the published LD, thus calling for efficient SPARQL query processing. An important category of queries that is prone to optimization is the ranking, or "top-k", queries, i.e. queries returning the top k results ordered by a specified ranking function.

Simple top-k queries can be expressed in SPARQL by appending to a SELECT query the ORDER BY and LIMIT clauses, which impose an order on the result set, and limit the number of results. Practitioners willing to issue top-k queries using complex ranking functions have been forced to create ad-hoc extensions such as project functions whose results can be used in the ORDER BY clause. This has lead to the inclusions of projection functions in the SPARQL 1.1

[9] working draft. Listing 1.1 provides an example of SPARQL 1.1 top-k query on a BSBM [3] dataset[3].

SPARQL engines supporting SPARQL and SPARQL 1.1 typically manage ORDER BY and LIMIT clauses are result modifiers that alter the solution generated in evaluating the WHERE clause before returning the result to the user. The semantics of modifiers imposes to take a solution as input, manipulate it, and generate a new solution as output. Specifically, an order modifier puts the solutions in the order required by the ordering clauses that are either ascending (indicated by ASC() that is also assumed as default) or descending (indicated by DESC()). The limit modifier defines an upper bound on the number of returned results; it allows to slice the result set and to retrieve just a portion of it. For instance, the query in Listing 1.1 is executed according to the query plan in Figure 1.a: solutions matching the WHERE clause are drawn iteratively from the RDF store until the whole result is materialized; then, the ordering function is evaluated monolithically, and the top 10 results are returned.

```
SELECT ?product ?offer ((?avgRateProduct + ?avgRateProducer) AS ?score)
WHERE {
 ?offer bsbm:product ?product .
 ?product bsbm:avgRate ?avgRateProduct ;
          bsbm:producer ?producer .
 ?producer bsbm:avgRate ?avgRateProducer.
}
ORDER BY DESC(?score)
LIMIT 10
```

Listing 1.1: "Example of a top-k query on BSBM"

As a result the performances of SPARQL top-k queries can be very poor when a SPARQL engine elaborates thousands of matching solutions and computes the ranking for each of them, even if only a limited number (e.g. ten) were requested. Moreover, the ranking predicates can be expensive to compute and, therefore, they should be evaluated only when needed and on the minimum possible number of results. It is clear that it may be beneficial in these cases to **split** the evaluation of the ranking projection function in ranking atoms, and **interleave** the evaluation of these ranking atoms with joins and boolean filters as shown in Figure 1.b.

Contribution. In a previous work [4], we presented a first sketch of $\mathcal{S}$PARQL-$\mathcal{R}$ANK algebra, and we applied it to the execution of top-k SPARQL queries on top of virtual RDF stores through query rewriting over a rank-aware RDBMS. In this paper, we propose a consolidated version of $\mathcal{S}$PARQL-$\mathcal{R}$ANK algebra and a general rank-aware execution model that can be applied to state-of-the-art SPARQL engine built on top of both RDBMS and native triple stores.

We provide an open source implementation of $\mathcal{S}$PARQL-$\mathcal{R}$ANK extending ARQ[4]) and we carry out some preliminary experiments.

Organization of the paper. In Section 2, we provide an introduction of SPARQL as presented in [15]. In Section 3, we show how we extended [15]

[3] For simplicity, we assume the average rates to be materialized in the dataset.
[4] The code is available at http://sparqlrank.search-computing.org/

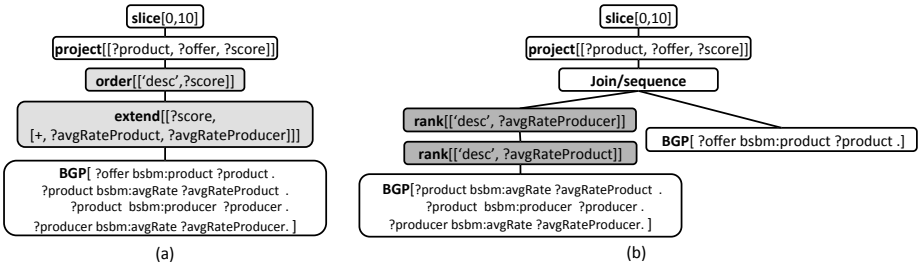

Fig. 1: Examples of (a) standard and (b) $\mathcal{S}$PARQL-$\mathcal{R}$ANK algebraic query plan for the top-k SPARQL query in Listing 1.1.

introducing a ranking model for SPARQL queries and proposing new algebraic operators of $\mathcal{S}$PARQL-$\mathcal{R}$ANK. In Section 5, we report on the preliminary results of the experiments we carried out comparing ARQ 2.8.9 with our rank-aware version. In Section 6, we present the related work. Finally, in Section 7, we elaborate on future works.

2 An Introduction to SPARQL Algebra

The features of SPARQL, taken one by one, are simple to describe and to understand. However, the combination of such features makes SPARQL a complex language whose semantics can only be fully understood through an algebraic representation. Several alternative algebraic models were proposed. Hereafter, we discuss the formalization presented in [15], focusing on the WHERE clause.

In SPARQL, the WHERE clause contains a set of *graph pattern* expressions that can be constructed using the operators OPTIONAL, UNION, FILTER and concatenation via a point symbol "." that means AND. Formally, a graph pattern expression is defined as:

Definition 1. *Assuming three pairwise disjoint sets I (IRIs), L (literals) and V (variables), a* ***graph pattern expression*** *is defined recursively as:*

1. *A tuple from* $(I \cup L \cup V) \times (I \cup V) \times (I \cup L \cup V)$ *is a graph pattern and in particular it is a triple pattern.*
2. *If* P_1 *and* P_2 *are graph patterns, then* $(P_1\ .\ P_2)$, $(P_1\ OPTIONAL\ P_2)$ *and* $(P_1\ UNION\ P_2)$ *are graph patterns.*
3. *If* P *is a graph pattern and* R *is a SPARQL built-in condition, then* $(P\ FILTER\ R)$ *is a graph pattern.*

A SPARQL built-in condition is composed by elements of the set $I \cup L \cup V$ *and constants, logical connectives* ($\neg, \wedge, \vee$)*, ordering symbols* ($<, \leq, \geq, >$)*, the equality symbol* ($=$)*, unary predicates like bound, isBlank, isIRI and other features.*

An important case of graph pattern expression is the Basic Graph Pattern:

Definition 2. *A **Basic Graph Pattern** (BGP) is a set of triple patterns that are connected by the "." (i.e., the AND) operator.*

The semantics of SPARQL queries uses as basic building block the notion of mapping that is defined as:

Definition 3. *Let P be a graph pattern, var(P) denotes the set of variables occurring in P. A **mapping** μ is a partial function $\mu : V \rightarrow (I \cup L \cup BN)$[5]. The domain of μ, denoted by dom(μ), is the subset of V where μ is defined.*

The **relation between** the notions of **mapping, triple pattern and basic graph pattern** is given in the following definition:

Definition 4. *Given a triple pattern t and a mapping μ such that $var(t) \subseteq dom(\mu)$, $\mu(t)$ is the triple obtained by replacing the variables in t according to μ. Given a basic graph pattern B and a mapping μ such that $var(B) \subseteq dom(\mu)$, we define $\mu(B) = \cup_{t \in B} \mu(t)$, i.e. $\mu(B)$ is the set of triples obtained by replacing the variables in the triples of B according to μ.*

Using these definitions, [15] defines the semantics of SPARQL queries as an algebra. The main algebra operators are Join ($\bowtie$), Union ($\cup$), Difference($\setminus$) and Left Join ($\bowtie$). The authors define the semantics of these operators on **sets of mappings** denoted with Ω. The evaluation of a SPARQL query is based on its translation into an algebraic tree composed of those algebraic operators.

The simplest case is the **evaluation of a basic graph pattern** defined as:

Definition 5. *Let G be an RDF graph and P a Basic Graph Pattern. The evaluation of P over G, denoted by $[\![P]\!]_G$, is defined by the set of mapping:*

$$[\![P]\!]_G = \{\mu |\ dom(\mu) = var(P) \text{ and } \mu(P) \subseteq G\}$$

If $\mu \in [\![P]\!]_G$, μ is said to be a solution for P in G.

The evaluation of more complex graph pattern is compositional and can be defined recursively from basic graph pattern evaluation by mapping the graph expressions to algebraic expressions.

Noteworthy, in SPARQL, the OPTIONAL and UNION operators can introduce unbound variables; it is known that the problem of verifying, given a graph pattern P and a variable $?x \in var(P)$, whether $?x$ is bound in P is undecidable [2], but an efficiently verifiable syntactical condition can be introduced. Hereafter, we propose such a syntactic notion of *certainly bound* variable, defined as:

Definition 6. *Let P, P_1 and P_2 be a graph patterns. Then the set of **certainly bound variables** in P, denoted as CB(P), is recursively defined as follows:*

1. *if t is a triple pattern and $P = t$, then CB(P) = var(t);*
2. *if $P = (P_1\ .\ P_2)$, then $CB(P) = CB(P_1) \cup CB(P_2)$;*

[5] BN is the set of blank nodes

3. *if* $P = (P_1\ UNION\ P_2)$, *then* $CB(P) = CB(P_1) \cap CB(P_2)$;
4. *if* $P = (P_1\ OPTIONAL\ P_2)$, *then* $CB(P) = CB(P_1)$;

The above definition recursively accumulates a set of variables that are certainly bound in a given graph pattern P because: they appear in graph pattern expressions that do not contain the OPTIONAL or UNION operators (rules 1 and 2), or they appear both on the left and on the right side of a graph pattern containing the UNION operator (rule 3), or they appear only in the left side of graph pattern expression that contains the OPTIONAL operator (rule 4)[6].

3 The $\mathcal{S}$PARQL-$\mathcal{R}$ANK Algebra

In this section, we progressively introduce: *a*) the basic concept of ranking criterion, scoring function and upper bound that characterised rank-aware data management [12], *b*) the concept of *ranked set of mappings*, an extension of the standard SPARQL definition of a set of mappings that embeds the notion of *ranking*, *c*) the new $\mathcal{S}$PARQL-$\mathcal{R}$ANK algebraic operators, and *d*) the new $\mathcal{S}$PARQL-$\mathcal{R}$ANK algebraic equivalences.

3.1 Basic Concepts

$\mathcal{S}$PARQL-$\mathcal{R}$ANK supports top-k SPARQL queries that have an ORDER BY clause that can be formulated as a scoring function combining several ranking criteria. Given a graph pattern P, a **ranking criterion** $b\colon \mathbb{R}^m \rightarrow \mathbb{R}$ is a function defined over a set of variables $?x_j \in var(P)$. The evaluation of a ranking criterion on a mapping μ, that is, the substitution of all of the variables $?x_j$ with the corresponding values from the mapping, is indicated by $b[\mu]$. A criterion b can be the result of the evaluation of any built-in function (having an arbitrary cost) of query variables.

A **scoring function** on P is an expression of the form $\mathcal{F}$ defined over the set B of ranking criteria. As typical in ranked queries, the scoring function $\mathcal{F}$ is assumed to be **monotonic**, i.e., a $\mathcal{F}$ for which holds $\mathcal{F}(x_1, \ldots, x_n) \geq \mathcal{F}(y_1, \ldots, y_n)$ when $\forall i : x_i \geq y_i$. In order for a scoring function to be evaluable, the variables in $var(P)$ that contribute in the evaluation of $\mathcal{F}$ must be bound. Since OPTIONAL and UNION clauses can introduce unbound variables, we assume all the variables in $var(P)$ to be *certainly bound*, i.e. variables that are certainly bound for every mapping produced by P (see also Definition 6 in Section 2). An extension of $\mathcal{S}$PARQL-$\mathcal{R}$ANK toward the relaxation of the *certainly bound variables* constraint is part of the future work and will be discussed in the conclusions of the paper.

Listing 1.1 provides an example of the scoring function $\mathcal{F}$ calculated over the ranking criteria *?avgRateProduct* and *?avgRateProducer*. We note that *?avgRateProduct* and *?avgRateProducer* are certain bound variables, as the query

[6] We omit discussing FILTER clauses since they cannot add any variable, granted that the variables occurring in a filter condition (P FILTER R) are a subset of $var(P)$.

contains no OPTIONAL or UNION clauses. The result of the evaluation is stored in the *?score* variable, which is later used in the ORDER BY clause.

Overall, a key property of $\mathcal{S}$PARQL-$\mathcal{R}$ANK is the ability to retrieve the first k results of a top-k query before scanning the complete set of mappings resulting from the evaluation of the WHERE clause. To enable such a property, the mappings progressively produced by each operator should flow in an order consistent with the final order, i.e., the order imposed by $\mathcal{F}$. When the evaluation of a SPARQL top-k query starts on the Basic Graph Patterns the resulting mappings are unordered. As soon as some $\mathcal{B} = \{b_1, \ldots, b_j\}$ (with $j < |B|$) of the ranking criteria can be computed (i.e., when $var(b_j) \subseteq dom(\mu)$), an order can be imposed to a set of mappings Ω by evaluating for each $\mu \in \Omega$ the ***upper bound*** of $\mathcal{F}[\mu]$ as:

$$\overline{\mathcal{F}}_{\mathcal{B}}[\mu] = \mathcal{F}\begin{pmatrix} b_i = b_i[\mu] & \text{if } b_i \in \mathcal{B} \\ b_i = max(b_i) & \text{otherwise} \end{pmatrix} \forall i$$

where $\max(b_i)$ is the application-specific maximal possible value for the ranking criterion b_i. $\overline{\mathcal{F}}_{\mathcal{B}}[\mu]$ is the upper bound of the score that μ can obtain, when $\mathcal{F}[\mu]$ is completely evaluated, by assuming that all the ranking criteria still to evaluate will return their maximal possible value. We can now formalize the notion of *ranked set of mappings*.

	?**p**	?**pr**	?**a1**	?**a2**	b_1	b_2	$\overline{\mathcal{F}}_{\{b_1 \cup b_2\}}$
μ_1	p1	pr3	4.0	4.5	0.80	0.90	1.70
μ_3	p3	pr4	2.0	3.5	0.40	0.70	1.10
μ_2	p2	pr2	2.0	3.0	0.40	0.60	1.00

Table 1: $\Omega'_{b_1} \bowtie \Omega''_{b_2}$

	?**p**	?**a1**	b_1	$\overline{\mathcal{F}}_{\{b_1\}}$
μ_1	p1	4.0	0.80	1.80
μ_2	p2	2.0	0.40	1.40
μ_3	p3	2.0	0.40	1.40

Table 2: Ω'_{b_1}

	?**pr**	?**a2**	b_2	$\overline{\mathcal{F}}_{\{b_2\}}$
μ_1	pr3	4.5	0.90	1.90
μ_3	pr4	3.5	0.70	1.70
μ_2	pr2	3.0	0.60	1.60

Table 3: Ω''_{b_2}

A **ranked set of mappings** $\Omega_{\mathcal{B}}$, with respect to a scoring function $\mathcal{F}$, and a set $\mathcal{B}$ of ranking criteria, is the set of mappings Ω augmented with an order relation $<_{\Omega_{\mathcal{B}}}$ defined over Ω, which orders mappings by their upper bound scores, i.e., $\forall\ \mu_1,\ \mu_2 \in \Omega :\ \mu_1 <_{\Omega_{\mathcal{B}}} \mu_2 \iff \overline{\mathcal{F}}_{\mathcal{B}}[\mu_1] < \overline{\mathcal{F}}_{\mathcal{B}}[\mu_2]$.

The monotonicity of $\mathcal{F}$ implies that $\overline{\mathcal{F}}_{\mathcal{B}}$ is always an upper bound of $\mathcal{F}$, i.e. $\overline{\mathcal{F}}_{\mathcal{B}}[\mu] \geq \mathcal{F}[\mu]$ for any mapping $\mu \in \Omega_{\mathcal{B}}$, thus guaranteeing that the order imposed by $\overline{\mathcal{F}}_{\mathcal{B}}$ is consistent with the order imposed by $\mathcal{F}$.

Note that a set of mappings on which no ranking criteria is evaluated ($\mathcal{B} = \emptyset$) is consistently denoted as $\Omega_\emptyset$ or simply Ω.

Table 1 depicts a subset of ranked set of mappings

$$\Omega_{\{?avgRateProduct, ?avgRateProducer\}}$$

(the ranking criteria are represented as b_1 and b_2 respectively) resulting from the evaluation of

$$\overline{\mathcal{F}}_{\{?avgRateProduct, ?avgRateProducer\}}$$

of the query in Listing 1.1, where mappings $\mu_i \in \Omega$ are ordered according to their upper bounds. When there are ties in the ordering, we assume an arbitrary deterministic tie-breaker function (e.g., by using the hash code of the lexical form of a mapping).

3.2 $\mathcal{S}$PARQL-$\mathcal{R}$ANK Algebraic Operators

Starting from the notion of *ranked set of mappings*, $\mathcal{S}$PARQL-$\mathcal{R}$ANK introduces a new *rank operator* ρ, representing the evaluation of a single ranking criterion, and redefines the Selection (σ), Join ($\bowtie$), Union ($\cup$), Difference($\setminus$) and Left Join ($\bowtie$) operators, enabling them to process and output ranked sets of mappings. For the sake of brevity, we present ρ and $\bowtie$, referring the reader to [4] for further details.

The *rank operator* ρ_b evaluates the ranking criterion $b \in B$ upon a ranked set of mappings $\Omega_{\mathcal{B}}$ and returns $\Omega_{\mathcal{B}\cup\{b\}}$, i.e. the same set ordered by $\overline{\mathcal{F}}_{\mathcal{B}\cup\{b\}}$. Thus, by definition $\rho_b(\Omega_{\mathcal{B}}) = \Omega_{\{\mathcal{B}\cup b\}}$. Tables 2 and 3 respectively exemplify the evaluation of $?avgRateProduct$ – to shorten, b_1 – an additional ranking criterion $?avgRateProduct2$ – b_2 – over the `?product bsbm:avgRate ?avgRateProduct` and `?product bsbm:avgRate ?avgRateProduct2` triple patterns. Moreover, the tables show the evaluation of the upper bounds $\overline{\mathcal{F}}_{\{b_1\}}$ and $\overline{\mathcal{F}}_{\{b_2\}}$.

The extended $\bowtie$ operator has a standard semantics for what it concerns the membership property [15], while it defines an order relation on its output mappings: given two ranked sets of mappings Ω'_{B_1} and Ω''_{B_2} ordered with respect to two sets of ranking criteria B_1 and B_2, the join between Ω'_{B_1} and Ω''_{B_2}, denoted as $\Omega'_{B_1} \bowtie \Omega''_{B_2}$, produces a ranked set of mappings ordered by $\overline{\mathcal{F}}_{B_1\cup B_2}$. Thus, formally $\Omega'_{B_1} \bowtie \Omega''_{B_2} \equiv (\Omega' \bowtie \Omega'')_{B_1\cup B_2}$. Table 1 exemplifies the application of the $\bowtie$ operator over the ranked set of mappings of Tables 2 and 3.

3.3 $\mathcal{S}$PARQL-$\mathcal{R}$ANK Algebraic Equivalences

Query optimization relies on algebraic equivalences in order to produce several equivalent formulations of a query. The $\mathcal{S}$PARQL-$\mathcal{R}$ANK algebra defines a set of algebraic equivalences that take into account the order property. The rank operator ρ can be pushed-down to impose an order to a set of mappings; such order can be then exploited to limit the number of mappings flowing through the physical execution plan, while allowing the production of the k results. In the following we focus on the equivalence laws that apply to the ρ and $\bowtie$ operators:

1. **Rank splitting** $[\Omega_{\{b_1,b_2,...,b_n\}} \equiv \rho_{b_1}(\rho_{b_2}(...(\rho_{b_n}(\Omega))...))]$: allows splitting the criteria of a scoring function into a series of rank operations $(\rho_{b_1}, ..., \rho_{b_n})$, thus enabling the individual processing of the ranking criteria.
2. **Rank commutative law** $[\rho_{b_1}(\rho_{b_2}(\Omega_B)) \equiv \rho_{b_2}(\rho_{b_1}(\Omega_B))]$: allows the commutativity of the ρ operand with itself, thus enabling query planning strategies that exploit optimal ordering of rank operators.

3. **Pushing ρ over $\bowtie$** [if $\Omega^{'}$ does not map all variables of the ranking criterion b, then $\rho_b(\Omega^{'}_{B_1} \bowtie \Omega^{''}_{B_2}) \equiv \rho_b(\Omega^{'}_{B_1}) \bowtie \Omega^{''}_{B_2}$; if both $\Omega^{'}$ and $\Omega^{''}$ map all variables of b, then $\rho_b(\Omega^{'}_{B_1} \bowtie \Omega^{''}_{B_2}) \equiv \rho_b(\Omega^{'}_{B_1}) \bowtie \rho_b(\Omega^{''}_{B_2})$]: this law handles swapping $\bowtie$ with ρ, thus allowing to push the rank operator only on the operands whose variables also appear in b.

The new algebraic laws lay the foundation for query optimization, as discussed in the following Section. We refer the reader to [4] for the complete set of equivalences.

4 Execution of Top-K SPARQL queries

In common SPARQL engines, a query execution plan is a tree of physical operators as iterators. During the execution of the query, mappings are drawn from the root operator, which draws mappings from underlying operators recursively, till the evaluation of a Basic Graph Pattern in the RDF store. The execution is incremental unless some blocking operator is present in the query execution plan (e.g., the ORDER BY operator in SPARQL).

In Section 3, we remove the logical barriers that make ranking a blocking operator in SPARQL. $\mathcal{S}$PARQL-$\mathcal{R}$ANK algebra allows for writing logic plans that split ranking and interleave the ranking operators with other operators evaluation. Thus, it allows for an incremental execution of top-k SPARQL queries. In the rest of the section, we first describe the $\mathcal{S}$PARQL-$\mathcal{R}$ANK incremental execution model and how to implement physical operators; then, we report on our initial investigations on a rank-aware optimizer that uses the new algebraic equivalences.

4.1 Incremental Execution Model and Physical Operators

The $\mathcal{S}$PARQL-$\mathcal{R}$ANK execution model handles ranking-aware query plans as follows:

1. physical operators incrementally output ranked sets of mappings in the order of the upper bound of their scores;
2. the execution stops when the requested number of mapping have been drawn from the root operator or no more mapping can be drawn.

In order to implement the proposed execution model, algorithms for the physical operators are needed. Some algorithms are trivial, e.g., selection that rejects solutions that do not satisfy the FILTER clauses while preserving the mapping ordering. For the non-trivial cases, e.g., ρ and $\bowtie$, many algorithms are described in the literature: MPro [7] and Upper [5] are two state-of-the-art algorithms useful for implementing the ρ operator, whereas the implementation of $\bowtie$ can be based on algorithms such as HRJN (hash rank-join) and NRJN (nested-loop rank-join) described in [13,11].

```
if Queue is not empty then
  if Queue.topscore()≥ Threshold then
    return Queue.top();
  end if
end if
loop
  if !I.hasNext() then
    Threshold ← 0.0;
  else
    mapping ← I.GetNext();
    Threshold ← mapping.score;
    insert mapping into Queue;
  end if
  if Queue is empty then
    break;
  end if
  if Queue.topscore()≥ Threshold then
    return Queue.top();
  end if
end loop
```

(a)

Q_{b_3}

	?pr	?of	b_1	b_2	b_3	$\overline{\mathcal{F}}_{b_1 \cup b_2 \cup b_3}$
μ_a	p1	o1	0.9	0.8	0.8	2.5
μ_b	p4	o3	0.7	0.7	0.9	2.3

Q_{b_2}

	?pr	?of	b_1	b_2	b_3	$\overline{\mathcal{F}}_{b_1 \cup b_2}$
μ_a	p1	o1	0.9	0.8	0.8	2.7
μ_b	p4	o3	0.7	0.7	0.9	2.4
μ_c	p1	o2	0.5	0.5	0.7	2.0

Ω_{b_1}

	?pr	?of	b_1	b_2	b_3	$\overline{\mathcal{F}}_{b_1}$
μ_a	p1	o1	0.9	0.8	0.8	2.9
μ_b	p4	o3	0.7	0.7	0.9	2.7
μ_c	p1	o2	0.5	0.5	0.7	2.5

(b)

Fig. 2: Example of the rank operator algorithm.

In Figure 2.a, we present the pseudo code of our implementation for the rank operator ρ. In particular, we show the `GetNext` method that allows a downstream operator to draw one mapping from the rank operator.

Let b be a scoring function not already evaluated (i.e., $b \notin B$). When $\mathcal{S}$PARQL-$\mathcal{R}$ANK applies ρ_b on a ranked set of mappings Ω_B flowing from an upstream operator, the drawn mappings from Ω_B are buffered in a priority queue, which maintains them ranked by $\overline{\mathcal{F}}_{B\cup\{b\}}$. The operator ρ_b cannot output immediately each drawn mapping, because one of the next mappings could obtain a higher score after evaluation. The operator can output the top ranked mapping of the queue μ, only when it draws from a upstream operator a mapping μ' such that

$$\overline{\mathcal{F}}_{B\cup\{b\}}[\mu] \geq \overline{\mathcal{F}}_{B}[\mu']$$

This implies that $\overline{\mathcal{F}}_{B\cup\{b\}}[\mu] \geq \overline{\mathcal{F}}_{B}[\mu'] \geq \overline{\mathcal{F}}_{B}[\mu'']$ for any future mapping μ'' and, moreover, $\overline{\mathcal{F}}_{B}[\mu''] \geq \overline{\mathcal{F}}_{B\cup\{b\}}[\mu'']$. None of the mappings μ'' that ρ_b will draw from Ω_B can achieve a better score than μ.

In Figure 2.b, we present an example execution of a pipeline consisting of two rank operators ρ_{b_3} and ρ_{b_2} that draws mappings from Ω_{b_1}. It is work to notice that the proposed algorithm concretely allows for splitting the evaluation of Ω_{b_1,b_2,b_3} in $\rho_{b_3}(\rho_{b_2}(\Omega_{b_1}))$ by applying the algebraic equivalence law in Proposition 1. Thus, it practically implements the intuition given in Figure 1.b.

When an operator downstream to ρ_{b_3} wants to draw a mapping from ρ_{b_3}, it calls the `GetNext` method of ρ_{b_3} that recursively calls the `GetNext` method of ρ_{b_2} that draws mapping ranked by $\overline{\mathcal{F}}_{b_1}$ from Ω_{b_1}. ρ_{b_2} has to draw μ_a and μ_b from Ω_{b_1}, before returning μ_a to ρ_3. At this point, ρ_3 cannot output μ_a yet, it needs

to call once more the `GetNext` method of ρ_{b_2}. After ρ_{b_2} draws μ_c from Ω_{b_1}, it can return μ_b that allows ρ_{b_3} to return μ_a.

5 Toward Rank-aware Optimization of Top-K queries

Optimization is a query processing activity devoted to the definition of an efficient execution plan for a given query. Many optimization techniques for SPARQL queries [18] exist, but none account for the introduction of the *ranking* logical property, which brings novel optimization dimensions. Although top-k query processing in rank-aware RDBMS is a very consolidate field of research, our investigations suggest us that existing approaches like [10] or [14] cannot be directly ported to SPARQL engines, as data in a RDF storage can be "schema-free", and, in some systems, it is possible to push the evaluations of BGP down to the storage system, a feature that is not present in RDBMS.

In order to devise query plans optimization for $\mathcal{S}$PARQL-$\mathcal{R}$ANK queries, some rank-aware optimizations must be advised. In this paper we focus on the *rank* operator, which is responsible for the ordering of mappings. We apply it within a naïve query plan that omits the usage of joins, thus losing the cardinality reduction brought by join selectivity; we just consider the evaluation of a single BGP, and the subsequent application of several rank operators to order mappings as they are incrementally extracted from the underlying storage system.

Notice that data can be retrieved from the source according to one ranking criterion b_i: in a previous work [4] we exploited a rank-aware RDBMS as a data storage layer offering indexes over ranking criteria. In such a case, additional ranking criterion are applied by serializing several rank operators. On the other hand, in this work we focus on native triple stores, namely, Jena TDB.

To have an initial assessment of the performance increase brought by the $\mathcal{S}$PARQL-$\mathcal{R}$ANK algebra with this naïve query plan, we extended the Jena ARQ 2.8.8 query engine with a new *rank* operator. We also extended the Berlin SPARQL Benchmark (BSBM), a synthetic dataset generator providing data resembling a real-life e-commerce website: we defined 12 test queries and, to exclude from the evaluation the time required for the run-time calculation of scoring functions, we materialized four numeric values for *Products*, *Producers* and *Offers*, each representing the result of a scoring function calculation.

Our experiments were conducted on an AMD 64bit processor with 2.66 GHz and 2 GB main memory, a Debian distribution with kernel 2.6.26-2, and Sun Java 1.6.0.

Table 4 reports the average execution time for the test queries, calculated for $k \in (1, 10, 100)$ on a 1M triple dataset. Notably, the performance boost of our prototype implementation with 1 variable queries (Q1, Q4, Q7, Q10) is at least one order of magnitude, regardless of optimizations. The good performances of the simple implementation techniques is justified by the co-occurence of the ranking function evaluation and sorting operation, which greatly reduce the number of calculation to be performed.

				SPARQL			SPARQL-RANK		
	Rank			ARQ			`Extended ARQ`		
Query	$\mathcal{F}$			**1**	**10**	**100**	**1**	**10**	**100**
Q1. Product	b_1			142	143	141	35	36	71
Q2. Product	b_1	b_3		255	256	244	126	364	381
Q3. Product	b_1	b_2	b_3	269	268	267	354	629	711
Q4. Product, Producer	b_2			173	170	170	45	47	171
Q5. Product, Producer	b_1	b_3		261	273	259	101	138	304
Q6. Product, Producer	b_1	b_2	b_3	295	293	293	300	388	612
Q7. Product, Offer	b_1			3863	3779	3854	467	461	948
Q8. Product, Offer	b_1	b_2		5705	5849	5847	907	936	1365
Q9. Product, Offer	b_1	b_2	b_3	6612	6485	6817	2933	5062	8933
Q10. Product, Producer, Offer	b_2			4026	4089	4055	509	520	494
Q11. Product, Producer, Offer	b_1	b_3		6360	6229	6359	1279	1337	1576
Q12. Product, Producer, Offer	b_1	b_2	b_4	8234	8165	8111	2304	3149	6137

Table 4: Query Execution Time for Dataset=1M and score functions $b_1 \rightarrow$ avgScore1 , $b_2 \rightarrow$ avgScore2, $b_3 \rightarrow$ numRevProd, $b_4 \rightarrow$ norm(price)

Table 4 also highlights queries where the performance of our prototype are comparable to ARQ. For instance, the poor (or worse) performance offered by Q2 and Q3 are due to the low correlation of the applied scoring functions, that, when split, require the system to perform several reordering on sets of ranked mappings. Finally, Q12 shows how the on-the-fly calculation of scoring predicates (b_4) still leads to better performance for our prototype.

This discussion calls for investigating more advanced, cost-based, optimization techniques that include join (or rank-join) operators, which can provide better performance boost due to join selectivity. Moreover, it would be interesting to try and estimate the correlation between the order of intermediate results imposed by multiple pipelined scoring functions evaluations. This is the subject of our future work. An extensive description of the settings and result of our experiment can be found at sparqlrank.search-computing.org, together with the latest results of this research work.

6 Related Work

Our work builds on the results of several well-established techniques for the efficient evaluation of top-k queries in relational databases such as [12,10,13,19] where efficient rank-aware operators are investigated, and [14] where a rank-aware relational algebra and the RankSQL DBMS are described.

The application of such results to SPARQL is not straightforward, as SPARQL and relational algebra have equivalent expressive power, while just a subset of the relational optimizations can be ported to SPARQL [18]. Moreover, the schema-free nature of RDF data demands dedicated random access data structures to

achieve efficient query evalutation; however, rank-aware operators typically rely on indexes for the sorted access; this can be expensive if naively done in native RDF stores, but cheaper in virtual RDF stores.

Our work contributes to the stream of investigations on SPARQL query optimization. Existing approaches focus on algebraic [15,18] or selectivity-based optimizations [21]. Despite an increasing need from practitioners [6], few works address SPARQL top-k query optimization.

Few works [8,20] extend the standard SPARQL algebra to allow the definition of ranking predicates, but, to the best of our knowledge, none addresses the problem of efficient evaluation of top-k queries in SPARQL. Straccia [22] describes an ontology mediated top-k information retrieval system over relational databases, where user queries are first rewritten into a set of conjunctive queries that are translated in SQL queries and executed on a rank-aware RDBMS [14]; then, the obtained results are merged into the final top-k answers. AnQL [24] is an extension of the SPARQL language and algebra able to address a wide variety of queries (including top-k ones) over annotated RDF graphs; our approach, instead, requires no annotations. Another rank-join algorithm, the Horizon based Ranked Join, is introduced [17] and aims at optimizing twig queries on weighted data graphs. In this case, results are ranked based on the underlying cost model, not based on an ad-hoc scoring function as in our work. The SemRank system [1] uses a rank-join algorithm to calculate the top-k most relevant paths from all the paths that connect two resources specified in the query. However, the application context of this algorithm is different from the one we presented, because it targets paths and ranks them by relevance using IR metrics. Moreover, the focus is not on query performance optimization.

7 Conclusion

In this paper, we presented $\mathcal{S}$PARQL-$\mathcal{R}$ANK, a rank-aware SPARQL algebra for the efficient evaluation of top-k queries. We introduced a new rank operator ρ, and extended the semantics of the other operators presented in [15]. To enable an incremental processing model, we added new algebraic equivalences laws that enable splitting ranking and interleaving it with other operators. In order to prototype an engine able to benefit from $\mathcal{S}$PARQL-$\mathcal{R}$ANK algebra, we extended both the algebra and the transformations of ARQ. We also run some preliminary experiments using our prototype on an extended version of the BSBM. The results show a significant performance gains when the limit k is in the order of tens, and hundreds of results.

As future work we plan to study additional optimizations techniques by, for instance, estimating the correlation between the order imposed by different scoring functions, and applying known algorithms to estimate the optimal order of execution of multiple rank operation obtained by splitting a complex ranking function. We also have preliminary positive results on a simple cost-base optimization techniques that uses rank-join algorithms [13,11] in combination with star-shaped patterns identification [23]. In addition, we plan to perform

an exhaustive comparison with the 2.8.9 version of the Jena ARQ query engine, which recently included an ad-hoc optimization for top-k queries, where the ORDER BY and LIMIT clauses are still evaluated after the completion of the other operations, but they are merged into a single operator with a priority queue that contains k ordered mappings. Finally, we outlook potential extensions of $\mathcal{S}$PARQL-$\mathcal{R}$ANK in dealing with SPARQL 1.1 federation extension and with the evaluation of SPARQL queries under OWL2QL entailment regime.

References

1. K. Anyanwu, A. Maduko, and A. Sheth. SemRank: ranking complex relationship search results on the semantic web. In *WWW '05*, pages 117–127. ACM, 2005.
2. C. B. Aranda, M. Arenas, and Ó. Corcho. Semantics and optimization of the sparql 1.1 federation extension. In *ESWC (2)*, volume 6644 of *Lecture Notes in Computer Science*, pages 1–15. Springer, 2011.
3. C. Bizer and A. Schultz. The Berlin SPARQL Benchmark. *Int. J. Semantic Web Inf. Syst.*, 5(2):1–24, 2009.
4. A. Bozzon, E. Della Valle, and S. Magliacane. Towards and efficient SPARQL top-k query execution in virtual RDF stores. In *5th International Workshop on Ranking in Databases (DBRANK 2011)*, August 2011.
5. N. Bruno, L. Gravano, and A. Marian. Evaluating Top-k Queries over Web-Accessible Databases. In *ICDE*, pages 369–. IEEE Computer Society, 2002.
6. P. Castagna. Avoid a total sort for order by + limit queries. JENA bug tracker. https://issues.apache.org/jira/browse/jena-89.
7. K. C.-C. Chang and S. won Hwang. Minimal probing: supporting expensive predicates for top-k queries. In *SIGMOD Conference*, pages 346–357. ACM, 2002.
8. J. Cheng, Z. M. Ma, and L. Yan. f-SPARQL: a flexible extension of SPARQL. In *DEXA'10*, DEXA'10, pages 487–494, 2010.
9. S. Harris and A. Seaborne. SPARQL 1.1 Working Draft. Technical report, W3C, 2011. http://www.w3.org/TR/sparql11-query/.
10. S.-w. Hwang and K. Chang. Probe minimization by schedule optimization: Supporting top-k queries with expensive predicates. *Knowledge and Data Engineering, IEEE Transactions on*, 19(5):646–662, 2007.
11. I. F. Ilyas, W. G. Aref, and A. K. Elmagarmid. Supporting Top-k Join Queries in Relational Databases. In *VLDB*, pages 754–765, 2003.
12. I. F. Ilyas, G. Beskales, and M. A. Soliman. A survey of top-k query processing techniques in relational database systems. *ACM Comput. Surv.*, 40(4), 2008.
13. I. F. Ilyas, R. Shah, W. G. Aref, J. S. Vitter, and A. K. Elmagarmid. Rank-aware Query Optimization. In *SIGMOD Conference*, pages 203–214. ACM, 2004.
14. C. Li, M. A. Soliman, K. C.-C. Chang, and I. F. Ilyas. RankSQL: query algebra and optimization for relational top-k queries. In *SIGMOD '05*, pages 131–142.
15. J. Pérez, M. Arenas, and C. Gutierrez. Semantics and complexity of SPARQL. *ACM Trans. Database Syst.*, 34(3), 2009.
16. E. Prud'hommeaux and A. Seaborne. SPARQL Query Language for RDF W3C Recommendation. http://www.w3.org/TR/rdf-sparql-query/, Jan. 2008.
17. Y. Qi, K. S. Candan, and M. L. Sapino. Sum-Max Monotonic Ranked Joins for Evaluating Top-K Twig Queries on Weighted Data Graphs. In *VLDB*, pages 507–518, 2007.

18. M. Schmidt, M. Meier, and G. Lausen. Foundations of SPARQL query optimization. In *ICDT '10*, pages 4–33, New York, NY, USA, 2010. ACM.
19. K. Schnaitter and N. Polyzotis. Optimal algorithms for evaluating rank joins in database systems. *ACM Transactions on Database Systems*, 35(1):1–47, 2010.
20. W. Siberski, J. Z. Pan, and U. Thaden. Querying the semantic web with preferences. In *ISWC*, pages 612–624, 2006.
21. M. Stocker, A. Seaborne, A. Bernstein, C. Kiefer, and D. Reynolds. SPARQL basic graph pattern optimization using selectivity estimation. In *WWW*, pages 595–604. ACM, 2008.
22. U. Straccia. SoftFacts: A top-k retrieval engine for ontology mediated access to relational databases. In *SMC*, pages 4115–4122. IEEE, 2010.
23. M.-E. Vidal, E. Ruckhaus, T. Lampo, A. Martínez, J. Sierra, and A. Polleres. Efficiently Joining Group Patterns in SPARQL Queries. In *ESWC (1)*, pages 228–242. Springer, 2010.
24. A. Zimmermann, N. Lopes, A. Polleres, and U. Straccia. A general framework for representing, reasoning and querying with annotated semantic web data. *CoRR*, abs/1103.1255, 2011.